Deterministic Service Guarantees for NAND Flash using Partial Block Cleaning

Siddharth Choudhuri
Center for Embedded Computer Systems
University of California, Irvine, USA
sid@cecs.uci.edu

Tony Givargis
Center for Embedded Computer Systems
University of California, Irvine, USA
givargis@uci.edu

ABSTRACT

NAND flash idiosyncrasies such as bulk erase and wear leveling results in non-linear and unpredictable read/write access times. In case of application domains such as streaming multimedia and real-time systems, a deterministic read/write access time is desired during design time. We propose a novel NAND flash translation layer called *GFTL* that guarantees fixed upper bounds (i.e., worst case service rates) for reads and writes that are comparable to a theoretical ideal case. Such guarantees are made possible by eliminating sources of non-determinism in GFTL design and using partial block cleaning. GFTL performs garbage collection in partial steps by dividing the garbage collection of a single block into several chunks, thereby interleaving and hiding the garbage collection latency while servicing requests.

Further, GFTL guarantees are independent of flash utilization, size or state. Along with theoretical bounds, benchmark results show the efficacy of our approach. Based on our experiments, GFTL requires an additional 16% of total blocks for flash management. GFTL service guarantees can be calculated from flash specifications. Thus, with GFTL, a designer can determine the service guarantees and size requirements apriori, during design time.

Categories and Subject Descriptors *D*.4.2 Operating Systems: Storage Management – *Secondary Storage, Allocation/deallocation strategies*
General Terms Design, Performance, Algorithms
Keywords NAND flash, Embedded Systems, Storage, QoS, Determinism, Real-Time, File Systems

1. INTRODUCTION

The proliferation of embedded systems has led to wide spread use of NAND flash as a storage medium. While the use of flash memory for secondary storage in mobile embedded systems has been known for over a decade [8], large scale adoption has only been possible recently due to affordable cost. With lowering cost per GB, NAND flash is poised to be used in newer application domains [12, 13]. For example, the One Laptop Per Child (OLPC) project, Canon's HD camcoder use NAND flash as the only non-volatile storage medium[15, 3]. While the economics of price has been favorable, the use of NAND flash in mission critical and real-time applications that demand determinism, has been a challenge due to NAND flash idiosyncrasies.

NAND flash has certain unique characteristics that are atypical of either RAM or hard disk drives. Specifically, NAND flash does not support in-place updates, i.e., an update (re-write) to a *page* (the minimum of write) is not possible, unless a larger region containing the page (known as a *block*) is first erased. Erase operation on a block is an order of magnitude slower, making it undesirable. Further, a block has a limited erase lifetime (typically 100,000) after which a block becomes unusable. Such characteristics require special handling of NAND flash by using either a dedicated file system or wrapping the NAND flash with a layer of hardware/software known as the *flash translation layer* (FTL). The FTL performs three important functions *(i)* Exports a view of NAND flash that resembles a disk drive, thereby hiding the peculiarities of NAND flash. Thus, an FTL translates a read/write request from the file system (*sector*) into a specific ⟨*block, page*⟩ of the NAND flash; *(ii)* Reclaims space by erasing obsoleted blocks (due to out of place updates), also known as *garbage collection*; *(iii)* Performs *wearleveling* to make sure that blocks across a flash get evenly erased.

NAND flash management (wearleveling, garbage collection) is workload dependent resulting in asymmetric read-write times. Therefore, typically FTLs do not provide service guarantees. For instance, consider a scenario in which an FTL is busy performing garbage collection over several blocks. During this time period I/O requests experience a high latency. Such latency might be tolerable for single-threaded, dedicated applications. However, as we move towards newer application domains, a deterministic service guarantee becomes desirable to design and run applications.

In this paper, we propose a NAND flash translation layer called as *GFTL* (for Guarantee Flash Translation Layer) that provides strict service guarantees for reads and writes that are close to an ideal case (to be described in Section 3). GFTL achieves this using a two fold approach. First, it uses a mapping from sectors to pages on flash that eliminates any dependency on flash utilization or state (i.e., provides determinism). Second, it uses partial block cleaning to hide the flash management latencies. Partial garbage collection is a scheme where the basic unit of garbage collection is a single block. Further, the garbage collection of each such block is divided into smaller states such that the garbage collection

CODES+ISSS'08, October 19–24, 2008, Atlanta, Georgia, USA.

and the file system read/write requests are *interleaved*, resulting in a responsive systems that hides garbage collection latency. The following are contributions of this paper:

1. An FTL that provides strict service time guarantees for reads and writes independent of the workload, utilization or the state of NAND flash.

2. Partial garbage collection, where the garbage collection of a single block is divided into chunks that are no greater than the largest non-interruptible flash operation; thereby providing a responsiveness that is close to a theoretical limit.

The rest of the paper is organized as follows. Section 2 describes the preliminaries of NAND flash. Section 3 presents the problem formulation followed by technical approach in Section 4. Section 5 describes our experimental setup and benchmarks. The results are provided in Section 6.

2. PRELIMINARIES

A NAND flash consists of multiple *erase blocks*. Each such erase block is further divided into multiple *pages*, a page being the minimum unit of data transfer (read/write). Associated with each page is a spare area known as the *Out Of Band (OOB)* area, primarily meant to store the Error Correction Code (ECC) of the corresponding page (also used to store meta-data such as inverse page table). A page is 512 bytes for older, small block NAND flash and 2 KB for newer large block NAND flash. Three basic operations can be performed on a NAND flash. An *erase* operation "wipes" an entire erase block turning every byte into all 1s i.e., `0xff`. A *write* operation works on either a page or an OOB area, selectively turning desired 1s into 0s. A *read* operation reads an entire page or an OOB area. Updates (re-writes) are out-of-place i.e., directed to a different page unless the entire block is erased. Table 1 depicts NAND flash specifications for the basic operations. There are two possible mappings

Table 1: NAND Flash Specifications

Characteristics	Samsung 16MB Small Block	Samsung 128MB Large Block
Block size	16384 (bytes)	65536 (bytes)
Page size	512 (bytes)	2048 (bytes)
OOB size	16 (bytes)	64 (bytes)
Read Page	36 (usec)	25 (usec)
Read OOB	10 (usec)	25 (usec)
Write Page	200 (usec)	300 (usec)
Write OOB	200 (usec)	300 (usec)
Erase	2000 (usec)	2000 (usec)

between a sector and a ⟨block, page⟩. A page based mapping where a translation table maps each sector to a ⟨block, page⟩ pair. However, the size of translation table can become a limiting factor as flash size increases. In order to deal with such a problem, a block based translation layer is widely used. For instance, in one of the popular block based translation layers known as NFTL [2], a sector is divided into a virtual block and an offset. The virtual block maps to a physical block (known as the primary block) on the NAND flash. In case of a rewrite (or if the primary block is full), a new physical block called a secondary block is chosen to perform the writes. When the two blocks become full, an operation known as fold merges the primary and the replacement blocks into a new primary block and freeing the old primary and replacement block. Garbage collection is invoked either when the NAND flash runs out of space (which does a fold across several blocks) or using a heuristic. Interested reader can find more details on mapping and garbage collection heuristics in [9, 4].

For the rest of the paper, the term flash refers to NAND flash. The following terminology is used throughout the paper: T_{wrpg} is the time to write a page and the OOB area; T_{rdpg} is the time to read a page; T_{rdoob} is the time to read an OOB area; T_{er} is the time erase a block; π is the number of pages per block; N is the total number of blocks on a flash; and L is the length of the write pending queue.

3. PROBLEM FORMULATION

We model an I/O request (incoming from file system to the FTL) as a real-time task $\tau = \{p, e, d\}$ where p is the periodicity, e is the execution time and d is the deadline. Without loss of generality, we assume that p is equal to d. We have two kinds of tasks: a read request task $\tau_r = \{p_r, e_r\}$, and a write request task $\tau_w = \{p_w, e_w\}$. p_r and p_w denote "how often" a read or write request arrives from the file system. e_r is the time taken to search for a given sector, read the corresponding ⟨*block, page*⟩ of the flash, and return a success/failure to the file system. Similarly, e_w is the time taken to write a sector to a given ⟨*block, page*⟩ . The bounds on p and e are determined by the FTL. Specifically, a *lower bound* on p (denoted by $\mathcal{L}(p)$) determines the maximum request arrival rate that an FTL can handle. The worst case execution time, i.e., an *upper bound* on e (denoted by $\mathcal{U}(e)$), determines the worst case rate at which requests are serviced by the FTL. For a file system, $\mathcal{U}(e)$ represents the *average memory access time* (AMAT) for read/write and $\mathcal{L}(p)$ represents the maximum rate at which requests are issued to the flash.

We now present a hypothetical ideal case that serves as a baseline for comparison. In an ideal case, the read/write access takes constant time. The bounds on $\mathcal{U}(e)$ for such an ideal case is shown in Table 2, i.e., there are no additional flash management overheads other than the actual page read/write [1] Note that, T_{er} is the longest *atomic* operation on a flash, i.e., when a block is being erased, the flash is locked and hence non-interruptible. Therefore, T_{er} is the limiting factor that decides the inter-arrival time (periodicity) of requests. Therefore, in an ideal case, $\mathcal{L}(p)$ is at least T_{er}. The latency due to T_{er} could be hidden by having buffers in the RAM. However, while this solution works for an average case, in a worst case scenario (i.e., when every access results in a block erase), one would require an infinitely large buffer in RAM as the arrival rate would exceed the service rate. This leads us to the following axiom: "*In the presence of flash management in a single chip flash, the block erase time T_{er} provides the lower bound on inter-arrival request time*". Although non-realtime block

Table 2: Service Guarantee Bounds

Bounds	Ideal	GFTL
$\mathcal{U}(e_w)$	T_{wrpg}	T_{wrpg}
$\mathcal{U}(e_r)$	$T_{rdpg} + T_{rdoob}$	$\pi T_{rdoob} + T_{rdpg}$
$\mathcal{L}(p_r)$ $\mathcal{L}(p_w)$	T_{er}	T_{er}+max$\{\mathcal{U}(e_w), \mathcal{U}(e_r)\}$

based FTLs like NFTL provide a write time close to T_{wrpg} in an average case, flash management results in a drastic, unpredictable variation. The motivation behind GFTL is to reduce this variation thereby enabling flash to be used in real-time applications. GFTL guarantees (Table 2) a worst case execution time for writes that is as good as an ideal case

[1]Without loss of generality and for simplicity, we exclude the flash controller execution time which is at least an order of magnitude less than flash access times (for microcontrollers used in FTLs)

and a worst case execution time for reads that is marginally ($(\pi - 1)T_{rdoob}$) larger than an ideal case. Further, GFTL provides service guarantees for requests that have an inter-arrival time [$\mathcal{L}(p)$] that is only slightly larger than an ideal case while performing garbage collection. The next section explains the technical details behind providing guarantees in Table 2.

4. TECHNICAL APPROACH

GFTL is a block based approach. A sector is treated as a *logical address* and a *logical block* is derived from the most significant bits of the logical address (Figure 1). A *block mapping table* is used to map a logical block to a physical block on the flash. For a given flash with N blocks, there is a $1:1$ mapping between the logical blocks and the physical blocks, resulting in N entries in the block mapping table. Q additional blocks are required to serve pending writes.

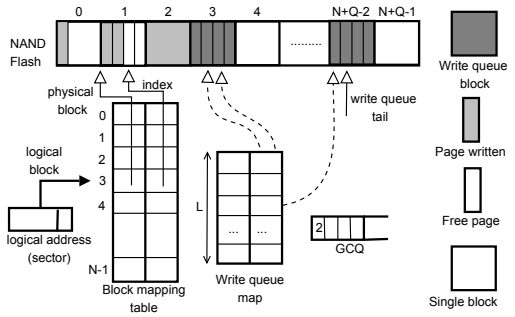

Figure 1: GFTL Data Structures

4.1 GFTL Writes

The first write to a given virtual block is written to a free physical block. Due to a $1:1$ mapping, a free physical block is guaranteed to be available. Once a physical block is found, pages are written sequentially starting from page 0. The sector number is written in the OOB area and serves as an inverse page table. After π writes, the physical block becomes full. The full physical block is added to a garbage collection queue called as *GCQ*. Additional writes that map to a full physical block are written to pages in the write queue (shown as dark gray in Figure 1). The write queue serves as a buffer for writes from the time a physical block becomes full until that physical block is garbage collected. A *write queue tail* serves as the index to the next available page in the write queue. There is only one write queue for the entire flash, thus, there exists a *write queue map* which maps the logical address (sector) to a $\langle block, page \rangle$ of the write queue.

A write either goes to the next available location pointed to by the index field of block mapping table (Figure 1) or into the write queue in case of a full physical block. In either case the time taken is constant i.e., T_{wrpg}. In case of a full block, the size of write queue is such that a page is guaranteed to be available. (shown in subsection 4.4.1). Thus, both the best and worst cast AMAT for writes is T_{wrpg}.

4.2 GFTL Reads

A read to a given sector is first searched in the write queue map since it holds the most recent copy. In case of a write queue map miss, the block mapping table is used to determine the physical block corresponding to the sector. The OOB area of the physical block is searched backwards A read from the write queue will result in one OOB read and one page read. A read from block mapping table on the other hand will result in π OOB reads in the worst case followed by the actual page read. Therefore, the best case AMAT for reads is $T_{rdpg} + T_{rdoob}$ and the worst case is $\pi T_{rdoob} + T_{rdpg}$.

4.3 GFTL Flash Management

The only flash management performed in GFTL is based on partial block cleaning which takes care of both garbage collection and wearleveling. The idea behind partial block cleaning is to perform garbage collection on a single block at a time. Further, each such single block garbage collection is divided into "partial" steps such that the time taken to perform each step is *no longer* than the longest atomic flash operation i.e., T_{er}. The partial steps are interleaved between servicing read/write requests. The garbage collection of a single block, say B_i, amounts to the following phases:

1. *Block Read:* In this phase, the pages that belong to B_i are first read from the write queue followed by reading the remaining valid pages out of the block B_i. In a worst case, this step can result in reading $(\pi - 1)$ pages from the write queue followed by π OOB reads of B_i to search the remaining valid page. Thus, the worst case time is $(2\pi - 1)T_{rdoob} + \pi T_{rdpg}$.
2. *Block Erase:* Block B_i is erased in time T_{er}.
3. *Block Write:* The pages that were read in phase 1 are written to a free block, say, B_{new}. In a worst case, π pages will be written resulting in a worst case time of πT_{wrpg}.

Since T_{er} is the longest atomic operation, we divide the block read and block write phases into partial steps, each of which is of a duration equal to T_{er} as shown in Figure 2(a). Let $\alpha = \lceil (2\pi - 1)T_{rdpg}/T_{er} \rceil$ denote the number of partial steps into which a read phase can be split as multiple of T_{er}. Similarly, $\beta = \lceil \pi T_{wrpg}/T_{er} \rceil$ denotes the number of partial steps that a block write can be broken into. Thus partial block cleaning divides the three block cleaning phases into $\kappa = (\alpha + 1 + \beta)$ steps, each of a duration T_{er}.

The core of GFTL acts as a real-time executive that implements the finite state machine shown in Figure 2(b). As shown in Figure 2(a), GFTL first dispatches any read/write request followed by performing a step of partial block cleaning (if the GCQ is non-empty). This approach lets GFTL provide read/write service guarantees shown in Table 2 while accepting requests at a rate equal to $\mathcal{L}(p)$. The wearlevel is automatically taken care by GFTL due to a round robin approach to allocating free blocks.

Over a period of time, blocks that belong to the write queue need to be erased. This is due to the fact that every page that belongs to a write queue block has been garbage collected. GFTL determines such blocks by scanning the write queue map. If a write queue block has no pointers pointing to it from the write queue map, the block is added to GCQ. The cost of garbage collecting a write queue block is only T_{er}.

4.4 Deterministic Guarantees

The read guarantees are implicit based on GFTL design. The write guarantees are based on an assumption that there is space readily available in the write queue when a physical block corresponding to the logical block becomes full. This assumption holds true if it can be shown that − in case of a worst case arrival sequence, the write queue growth is finite and can be determined apriori.

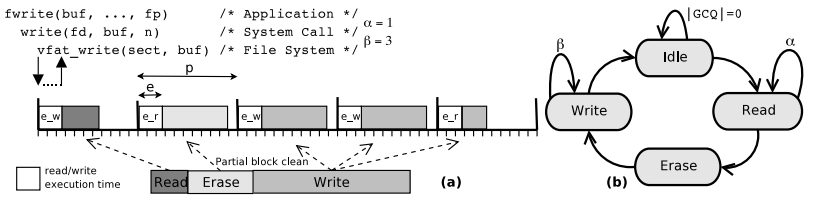

Figure 2: Partial Block Cleaning and FSM

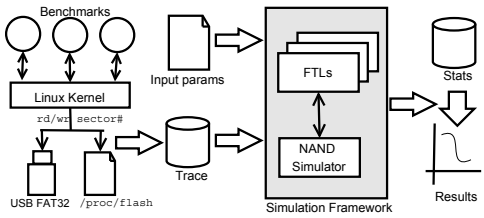

Figure 3: Setup

The following is a worst case scenario: $N \times \pi$ write requests arrive making all N blocks full. Subsequent write requests arrive in a sequence (filling up the write queue) such that each request belongs to a different logical block, and two requests that belong to the same logical block are spread out as far apart as possible in the write queue, i.e., separated by a distance of N. Therefore the worst case sequence of logical blocks to which writes arrive are $\{0, 1, 2, ..., N-1, 0, 1, 2, ..., N-1, ...\}$ (Figure 4 "Block Numbers Arrival Sequence"). This results in each write queue block being filled with π pending writes, each of which belongs to a unique logical block. Therefore, a write queue block cannot be reclaimed until π blocks are first garbage collected (i.e., worst case for a write queue block). Since each incoming write request is a pending write, the write queue grows at a rate equal to $1/\mathcal{L}(p)$. Every $\kappa \times \mathcal{L}(p)$ time units (where $\kappa = \alpha + \beta + 1$), a block is garbage collected (Figure 4 "Service Rate"). Each block that is garbage collected also renders the page(s) in write queue (which belong to the logical block) obsolete. Since the arrival rate, $1/\mathcal{L}(p)$, is greater than the service rate, $1/(\kappa \times \mathcal{L}(p))$, theoretically this leads to an infinite queue length (Figure 4 "Theoretical Write Queue Length"). However, in our worst case arrival model, after first N writes, every incoming write request already has at least one or more pending write(s) in the write queue that belongs to the same logical block.

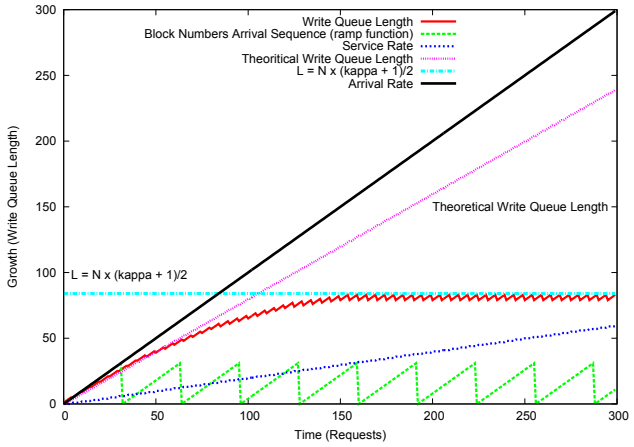

Figure 4: Write Queue Length Growth

Specifically, after first N request, every incoming write request already has one page in the write queue from the past, that belongs to the same logical block as the incoming write. Similarly, after $2N$ writes, every write request has 2 pending requests and so on. Thus, with time, the growth of the write queue length decreases every N requests reaching a steady state value (Figure 4 "Write Queue Length"). It can be proven that the growth of the write queue length is bounded by $max(L) = N \times (\kappa+1)/2$, where $\kappa = \alpha+\beta+1$ in the worst case arrival sequence (Figure 4) "L"). The details of proof for such a bound is in [7].

Thus, the write queue length can be calculated at design time by looking at the flash specs. Note that though L is greater than N (total blocks), the actual write queue length in terms of the number of additional blocks is $[N \times (\kappa + 1)/2]/\pi$ as each block can store π pending writes.

5. EXPERIMENTAL SETUP

Figure 3 shows our experimental setup. A USB flash disk, formatted as a FAT 32 file system was connected to a PC running Linux kernel 2.6.16. The kernel was modified to sniff low level file read/write requests being issued to the USB flash and log the requests (sector, read/write operation) into `/proc/flash`. A series of benchmarks were run to generate trace data. The trace, along with input parameters (block size, page size, etc) is fed to our simulation framework.

We used the following benchmarks representing a variety of workloads. The *Andrew* benchmark [10] consists of five phases involving creating files, copying files, searching files, reading every byte of a file and compiling source files. The *Postmark* benchmark measures performance of file systems running networked applications like e-mail, news server and e-commerce [11]. The *iozone* benchmark [14] is a well known synthetic benchmark. We ran iozone to do read, write, rewrite, reread, random read, random write, backward read, record rewrite and stride read on file sizes ranged from 64KB to 32MB in strides of 2×. Besides these standard benchmarks, we used our own benchmark called *consumer*. The consumer benchmark simulates flash activities commonly used in consumer electronics devices such as image manipulation, data transfer, audio and video playback.

A set of benchmarks were run in sequence to generate a file system *trace*. The first trace, called the *synthetic* trace was generated by running the following sequence: format flash → andrew → postmark → iozone. Similarly, consumer trace was generated by formating a flash followed by running the consumer benchmark. In order to perform a rigorous evaluation of GFTL, each read/write in the trace was simulated with a periodicity of $\mathcal{L}(p)$ i.e., there is *no idle period*. Further, the synthetic trace consists of 4.3 million writes and 27,841 reads and the consumer trace consists of 125,596 writes and 76,479 reads. The flash size at 100% utilization for synthetic trace is 136 MB and 260 MB for the consumer trace. The simulations are based on data sheet values for large page flash (Table 1). Details on benchmark characteristics can be found in [6].

6. RESULTS

Table 3 depicts a summary of runs for the two traces based on varying utilization and pages per block (π). The first two columns under $\bar{e}_{wr}$ and $\bar{e}_{rd}$ denote the average write

Table 3: GFTL Performance

	%	π	$\bar{e}_{wr}$ usec	e^{max}_{wr} usec	σ^{e}_{wr}	$\bar{e}_{rd}$ usec	e^{max}_{rd} usec	σ^{e}_{rd}	w_i	Δ %
Synthetic	50	16	244	300	116	231	425	119	0.43	16
		32	217	300	133	428	825	239	1.11	14
		64	212	300	136	789	1625	474	3.05	14
	100	16	244	300	116	231	425	119	3.29	16
		32	217	300	133	428	825	239	8.17	14
		64	212	300	136	789	1625	474	14.2	14
Consumer	50	16	299	300	11	237	115	425	0.00	16
		32	299	300	13	237	425	115	0.01	13
		64	299	300	14	836	1625	462	0.01	12
	100	16	299	300	11	237	425	115	0.01	16
		32	299	300	13	437	825	231	0.02	13
		64	299	300	14	836	1625	462	0.02	12

and read access times (AMAT) for each run of the trace. Similarly, e^{max}_{wr} and e^{min}_{rd} denote the maximum and minimum recorded AMAT. The standard deviation is denoted by σ^{e}_{wr} for writes and σ^{e}_{rd} for reads. The effectiveness of wearleveling is measured using *wear index* denoted by w_i. wear index is a quantitative measure of wearlevel calculated as $(\sigma_{erase}/N) \times 100$, where σ_{erase} is the standard deviation of the number of erases per block. σ_{erase} takes into account the "variation" in number of times a block is erased. To take into account the size of flash in determining wearlevel, we used σ_{erase}/N as an indicator of wearlevel.GFTL incurs overhead due to the write queue. This overhead is measured as the percentage increase in flash size denoted by column entitled Δ in Table 3. The following observations are made based on Table 3: *(i)* As mentioned in Table 2, the maximum write time is equal to T_{wrpg}. The average write time is less than T_{wrpg} because some of the writes occur during the FSM state change resulting in being written to a block buffer (written later on to the flash and accounted in partial garbage collection time). The maximum read time depends on pages per block for a given flash. This is due to the fact that larger π implies longer chain of OOBs to read. The maximum values observed are equal to $\pi T_{rdoob}+T_{rdpg}$ which is equivalent to searching the entire block for a given page. The standard deviation for reads also shows a similar trend i.e., increasing with larger π *(ii)* The average read/write service times ($\bar{e_{rd}}/\bar{e_{wr}}$) are independent of the flash utilization. This result departs from conventional approaches such as NFTL where the AMAT varies as the flash utilization increases. The standard deviation is higher in case of the synthetic trace because of the rewrites and rereads made to the flash by the iozone benchmark. Note that if a block is in the middle of partial garbage collection, additional reads and writes are serviced by the in memory block buffer resulting in an almost zero service time. This leads to the high variation in the average values. *(iii)* The wearlevel index depends on the flash utilization and the number of pages per block, π. As the flash utilization increases, blocks get recycled more often. However, blocks that are read-only are not erased leading to the large gap between minimum and maximum values. This shows as an increase in the wear index. *(iv)* The value of Δ reflects the additional blocks used by GFTL to maintain the write queue. This value was calculated (and used in simulation) based on the equation $L = N \times (\kappa+1)/2$. Given the lowering cost per GB of NAND flash, such an overhead is tolerable.

Figure 6 shows the distribution of execution time and partial GC in a given period. The total length of a histogram represents a single period i.e., $\mathcal{L}(p)$. The "Rd/Wr request"

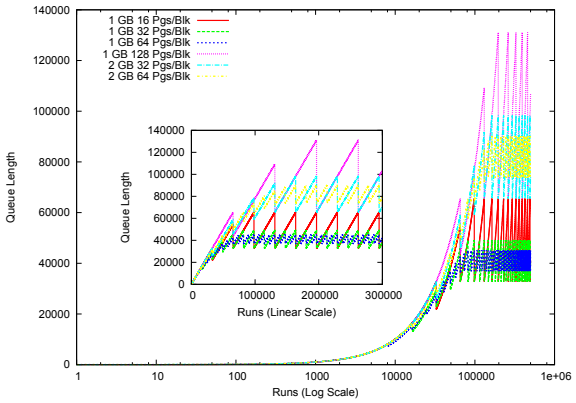

Figure 5: Write Queue Length Growth Simulation

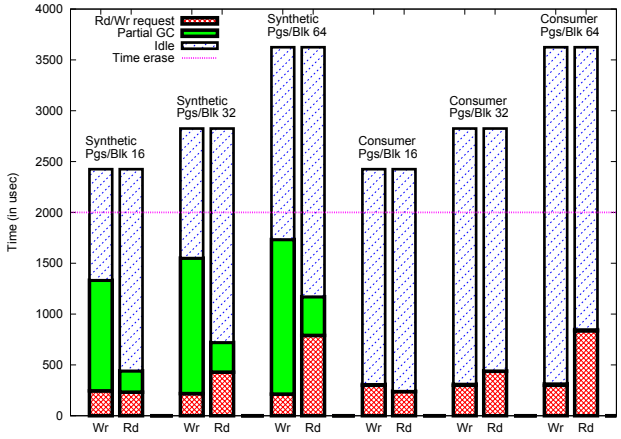

Figure 6: Average Time Distribution per Period

bar denotes the average service time i.e., $\bar{e}$. The remaining time is spent doing either partial GC or being idle (i.e., when GC queue is empty). Though GFTL guarantees an arrival rate equal to the length of the histogram, a fraction of partial GC time is spent idle because the guarantees are calculated based on worst case scenario. For the given traces, the sum of service time and partial GC is less than the T_{er}. This idle time decreases with increasing π as the value of κ increases which implies that the number of states in the FSM (Figure 2) also increases leading to longer time spent in GC. Note that in case of the consumer benchmark, the amount of time spent in performing partial GC is negligible compared to $\mathcal{L}(p)$. Though the consumer trace represents a larger flash size the number of read/writes performed is less than the synthetic trace. The efficacy of GFTL is shown by keeping the flash busy for time that is close to the largest non-interruptible time, T_{er} (Figure 6) while still giving applications a service time that is close to ideal and independent of the flash utilization (Table 3).

Figure 5 analyzes the overhead of GFTL in terms of write queue length. The x-axis is the number of runs simulated and the y-axis shows the growth of write queue length L. Initially, the write queue grows at a rate that is close to the incoming request rate and after every N requests, the slope decreases to a final "steady state". During this state, the write queue length varies depending on the value of N and π. Figure 5 shows the growth for a specific large page flash from Table 1. The growth depends on the values of T_{er}, T_{wrpg} and T_{rdpg}. Specifically, the larger the difference between the block erase time and the block read/write

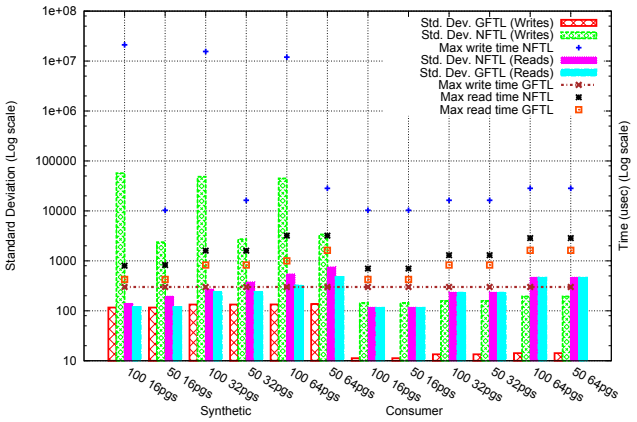

Figure 7: NFTL vs. GFTL

time, the larger the value of L as GFTL would require more states.Figure 7 compares GFTL with NFTL. The variation in write times is more than an order of magnitude less for GFTL due to deterministic guarantees. The only standard deviation in GFTL is due to some of the writes going into an in memory buffer during the times of partial block cleaning. The maximum write time of GFTL is constant (as opposed to NFTL) and the maximum read time is proportional to π.

The GFTL RAM overhead over traditional block based FTLs is largely due to the write queue map. Specifically, the write queue map takes $L \times 32$ bytes. For a 1 GB, 32 pages/block flash, the size of write queue map is 1.5 MB. In terms of reliability, in case of a power failure, the required mapping structures can be recreated by scanning flash OOB during startup. In terms of scalability, we ran GFTL for a flash size of over 1 GB on iozone benchmark with similar results and adherence to bounds from Table 2.

7. RELATED WORK

While there have been several FTLs, the real time aspect of NAND flash was first investigated by [5]. The authors proposed an innovative approach towards using a garbage collector thread (instance) for each real time task. The garbage collector thread has a execution time of $(\pi - \alpha) \times (T_{rdpg} + T_{wrpg}) + T_{er}$+cpu_time). Each garbage collector invocation takes at least $(\pi-1)(T_{rdpg}+T_{wrpg})+T_{er})$ time (ignoring cpu time) in the *best case*. In our approach, the overhead of partial GC is T_{er} in the *worst case*. Moreover, with GFTL we do not associate an additional GC task thereby avoiding overhead. [5] requires file system support for special `ioctl` calls. GFTL can be run on top any unmodified file system. Results from [5] are based on two tasks $T1 = (3, 20)$ and $T2 = (5, 20)$ resulting in creation of two GC tasks $G1 = (22, 160)$ and $G2 = (22, 600)$ at 50% utilization. The execution time of GC thread is comparable to 10 times T_{er}. GFTL on the other hand provides a delay that is around T_{er}. Moreover, we provide a rigorous evaluation where each request is considered a real-time task along with high utilization.

In [1], the authors address soft real-time issues by modifying the file system; the focus being commonly used access patterns and not strict guarantees. In [9], the authors survey a wide range of garbage collection algorithms as part of their study. However, the garbage collectors are not aimed at real-time systems. An exhaustive research on flash memories for real time systems was done by [16]. The conclusions in [16], supports our motivation for the lack of real-time, deterministic guarantees for flash.

8. CONCLUSION

In this paper we presented GFTL which provides $O(1)$ write time and a read time that takes π (pages per block) searches of the flash OOB in the worst case. Partial block cleaning lets requests arrive at a rate comparable to T_{er}, the block erase time which is a theoretical limit on responsiveness of a flash. Benchmark results show that GFTL sticks to the theoretical limits independent of the flash utilization or state. GFTL lets a developer calculate the service guarantees and size requirements from the flash specifications during design time. The flash overhead from experiments is 16% on average. The comparisons made against NFTL (with increased size) and previous work on real-time garbage collection show the efficacy of our approach. In summary, GFTL enables a highly responsive flash with strict guarantees. In our future work, we will look at ways to reduce block erases performed by GFTL and the power consumption aspects.

9. ACKNOWLEDGMENT

This work was in part supported by grant #0749508 from the National Science Foundation.

10. REFERENCES

[1] New techniques for real-time FAT file system in mobile multimedia devices. *IEEE Transactions on Consumer Electronics*, 52:1–9, 2006.
[2] A. Ban. Flash file system optimized for page-mode flash technologies. *US Patent 5,937,425*, Aug 10, 1999.
[3] Canon. Vixia HD Camcoder, January 2008.
[4] L.-P. Chang and T.-W. Kuo. Efficient management for large-scale flash-memory storage systems with resource conservation. *Trans. Storage*, 1(4):381–418, 2005.
[5] L.-P. Chang, T.-W. Kuo, and S.-W. Lo. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *TECS*, 3(4):837–863, 2004.
[6] S. Choudhuri and T. Givargis. Performance improvement of block based NAND flash translation layer. In *IEEE/ACM CODES+ISSS '07*, pages 257–262, New York, NY, USA, 2007. ACM.
[7] S. Choudhuri and T. Givargis. Real-time access guarantees for NAND flash using partial block cleaning. In *SEUS '08*. Springer Verlag LNCS, 2008.
[8] F. Douglis, R. Caceres, M. F. Kaashoek, K. Li, B. Marsh, and J. A. Tauber. Storage alternatives for mobile computers. In *OSDI*, pages 25–37, 1994.
[9] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comp. Surv.*, 37(2):138–163, 2005.
[10] J. H. Howard and et. al. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, 1988.
[11] J. Katcher. Postmark: A new file system benchmark. Technical report, Net App. Inc, TR 3022, 1997.
[12] G. Lawton. Improved flash memory grows in popularity. *Computer*, 39(1):16–18, 2006.
[13] MemCon. MemCon, July 2007. `http://linuxdevices.com/news/NS6633183518.html`.
[14] W. Norcutt. IOZONE benchmark, `www.iozone.org`.
[15] One Laptop Per Child Project. `http://laptop.org`.
[16] D. Parthey. Analyzing real-time behavior of flash memories. *Diploma Thesis, Chemnitz University of Technology*, April, 2007.